Format Preserving Encryption Enhancement to PyCryptodome*

Enabling FPE for the masses via popular cryptographic library

Joshua Holt

Georgia Institute of Technology, jholt@gatech.edu

Capstone Project

Format Preserving Encryption (FPE) is a method of encryption which encrypts a plaintext into a ciphertext while preserving the format of the plaintext. For example, if the plaintext is a social security number, the resulting ciphertext will also be a nine-digit decimal number. This preservation can apply to credit card numbers, postal addresses, networking payloads (including IP addresses), names of people and places, and can be used to provide encryption and stateless tokenization of sensitive data.

While FPE has been widely discussed in many academic papers, and the subject of several patent claims, it has not yet been integrated into popular cryptographic libraries. This paper outlines our implementation of FF3-1, recommended by NIST 800-38G into PyCryptodome, a widely used cryptographic library in the Python ecosystem.

CCS CONCEPTS • Security and privacy • Cryptography • Symmetric cryptography and hash functions • Block and stream ciphers

1 INTRODUCTION

Strong encryption schemes can guarantee confidentiality, and when used appropriately, can effectively prevent disclosure and compromise of sensitive information. However, block ciphers, which provide the backbone of symmetric cryptography, are not always the best tools to utilize when either the length or the format of the data needs to be preserved.

Format-preserving encryption, as the name implies, maintains both the format and length of the original plaintext. FPE is useful for preserving the confidentiality of personally identifiable information, including social security numbers, driver licenses, U.S. passport numbers, IP addresses, names, street addresses and email addresses, as well as sensitive financial information such as credit card and bank account numbers. It is ideal for legacy applications and databases which may be infeasible or costly to retrofit with modern cryptographic support. FPE can act as a stateless tokenization solution to limit or eliminate compliance requirements for merchants and service providers under the Payment Council Industry Data Security Standard [18, 19], and can provide pseudonymization while maintaining the structure of a data type [16].

The author provides the following contributions: 1) a reference implementation of the FF3-1 format-preserving encryption mode of operation, as outlined in NIST 800-38G [1], 2) integration with the popular PyCryptodome cryptographic library, 3) validation with the NIST Automated Cryptographic Validation Program test vectors [21], 4) documentation providing API operation, use cases, and best practices and 4) release under the BSD-2 clause license [31], allowing “free-as-in-speech" usage rights.

While other implementations of format-preserving encryption (including FF3-1) exist, to date the mode of operation has not been integrated with any major open-source cryptographic library. The author has chosen to enhance

PyCryptodome, a fork of the PyCrypto library, with the goal of providing a simplified API, quality documentation, and working examples. Code examples and documentation are "the stronger predictors of production working code", while "simplified APIs...promote better security results" [17]. By providing a reference implementation as part of a trusted cryptographic library with mechanisms for key management, random number generation, and other cryptographic features, developers can avoid rolling their own cryptographic solutions or evaluating standalone implementations. Finally, the implementation serves as a reference for developers and aspiring cryptographers, as it closely follows the NIST specification.

This paper is organized as follows: Section 2 provides background information on format-preserving encryption including industry and open-source implementations. Section 3 details the FF3-1 algorithm and implementation within PyCryptodome, while Section 4 compares the performance of the implementation. The threat model for format-preserving encryption and FF3-1 is considered in Section 5, with use cases and recommendations for FPE summarized in Section 6. The complete implementation and documentation are provided as an appendix.

2 BACKGROUND OF FORMAT PRESERVING ENCRYPTION

2.1 ACADEMIC BACKGROUND AND RELATED WORK

The concept of a tweakable block cipher was first proposed by Liskov et al. [13]. As block ciphers are "inherently deterministic every encryption of a given message with a given key will be the same" [13]. The authors proposed a second input, known as the "tweak." The goal was to distinguish the function of the tweak, which provides variability, from that of the key, which provides uncertainty, and is designed in such a way that modifying the tweak is "less costly than changing the key" [13]. The first formal treatment of format-preserving encryption was provided in [2], providing FPE constructions, syntax, and security notions.

The FFX mode of operation for FPE was specified in [4], which uses Feistel-based encryption with multiple sets of parameters including tweak values. A Feistel construction is a "natural approach for solving the small-space FPE problem" as it is "simple, old, and extensively studied" [3]. FFX "can encipher strings of any length over any desired alphabet" depending on the parameter set used; parameter set A2 encrypts binary strings, while A10 allows for encryption of decimal strings [3]. FFX[radix] is introduced in [5] which "effectively unifies and extends FFX-A2 and FFX-A10" allowing for arbitrary message lengths and a constant number of Feistel rounds.

The BPS format-preserving encryption proposal was outlined in [6], capable of encrypting "short or long strings composed of characters from any set" along with a tweak capability and the ability to "use any standardized primitives such as TDES, AES, or SHA-2." The authors outline two contributions. The first is the introduction of the internal block cipher BPS-BC (along with the inverse BC^{-1}) based on a Feistel structure that can encrypt and decrypt values of limited length. The number of Feistel rounds is fixed at 8, and the tweak is also a fixed length (64 bytes). The second contribution is the BPS mode of operation combining the BC cipher in a manner "similar to the classical Cipher-Block Chaining mode" to allow format-preserving encryption of inputs up to 8MB in size.

The NIST Special Publication 800-38G published in 2016 provided "approved methods of format-preserving encryption" along with algorithms, parameter choices and security goals [27]. The FFX[radix] parameter set and the BPS-BC mode of operation was submitted to NIST and adopted as FF1 and FF3, respectively. A second mode of operation (FF2) was submitted but not approved due to security weaknesses. The NIST publication provided example values for implementations of the provided algorithms [20] with the intention of allowing "conformance testing for implementations...within the framework of the Cryptographic Algorithm Validation Program (CAVP)" [22]. A draft revision of 800-38G, released in 2019, increased the minimum domain size and modified the tweak structure for FF3 due to published cryptographic vulnerabilities [1]. A fourth mode of operation (FF4), also known as DFF ("delegatable FF") has recently been submitted to NIST for inclusion. This mode is based on FF2 but modifies the Feistel round function to circumvent the security weaknesses [29].

Identity-based tweakable block ciphers were introduced in [28]. The aim of this construction is to provide privacy guarantees by using a Password-Based Key Derivation Function to ensure that even if an individual encryption key is disclosed, it does not affect other encryptions under the same base key. The standard FF1 and FF3 schemes are not eligible, making the potential inclusion of FF4/DFF within the NIST approved modes of operation an interesting area to watch.

The FF1 and FF3 modes of operation utilize the AES block cipher, which enjoys hardware acceleration on processors with AES-NI instruction sets. By replacing the AES block cipher with lightweight block ciphers (LEA and SPECK), efficiency can be improved in Internet-of-Things (IOT) devices [30].

2.2 IMPLEMENTATIONS OF FORMAT-PRESERVING ENCRYPTION

Industry implementations have the potential advantage of NIST validation and FIPS compliance but may require substantial hardware and licensing costs. Individual implementations of FPE may not be compatible, making it difficult to migrate or replace (vendor lock-in). Implementations may use unapproved modes of operation (FF2) and deprecated block ciphers (3DES). Finally, the underlying algorithms and block ciphers might not be disclosed, leaving the customer unable to validate the security properties of the solution. In all cases surveyed, the pricing of the proprietary industry implementations is not disclosed without contacting sales, requiring potentially considerable capital and operating expense.

Table 1: Survey of industry implementations of Format-Preserving Encryption

Vendor	Product	Modes of Operation	Cost
Hashicorp	Vault Advanced Data Protection	FF3-1 with AES	Contact Sales
IBM	Common Cryptographic Architecture	FF1, FF2, FF2.1 with 3DES	Contact Sales
Micro Focus	Voltage SecureData	FF1 with AES	Contact Sales
BlueFin	ShieldConex	Unknown FPE algorithm	Contact Sales

Several standalone open-source implementations of format-preserving encryption schemes have been released. CapitalOne provided an original implementation in the Go programming language. Recently, Mysto has provided FPE implementations in several programming languages.

Table 2: Survey of open-source implementations of Format-Preserving Encryption

Author	Programming languages supported	Algorithm	License
CapitalOne	Go	FF1, FF3	Apache 2.0
Mysto	Python, Java, C, Node	FF3	Apache 2.0
Kpdyer (LibFFX)	Python	FF1	GNU GPL 3.0
Emulbreh (PyFFX)	Python	FFX-A2	MIT

Open-source implementations have not been integrated with common cryptographic frameworks (examples including PyCryptodome, Pyca/Cryptography, LibSodium and the Go Supplementary Cryptographic Libraries). The exception is BouncyCastle, which has released format-preserving encryption with version 1.69 but is limited to the Java ecosystem.

In addition, the FF1/FFX algorithm is patented by Voltage Systems [23]. While a letter of assurance for essential patent claims was provided to NIST [24], the licensing costs make the FF1 mode of operation infeasible for inclusion in an open-source cryptographic library. Due to the patent claims, the author did not consider FF1 a valid implementation option.

3 THE FF3-1 ALGORITHM AND IMPLEMENTATION WITHIN PYCRYPTODOME

3.1 THE FF3-1 ALGORITHM AND PRELIMINARIES

The FF3-1 (Format-preserving, Feistel-based, 3rd submission, 1st revision) is a format-preserving encryption scheme [1]. It is a modified version of the BPS-BC cipher originally outlined in [6], and provides two functions:

- FF3-1.Encrypt(K, T, X): Takes as input key K, tweak bit string T, and numeral string X, and returns a numeral string Y as ciphertext
- FF3-1.Decrypt(K, T, X): Takes as input key K, tweak bit string T, and ciphertext numeral string X, and returns a numeral string Y as plaintext

Note the scheme provides "correct decryption [26]", the output of FF3-1.Encrypt, when passed with the same key and tweak to the input of FF3-1.Decrypt, will provide the original plaintext.

The key is formally listed as "an input to the encryption and decryption functions [1]" but is simply an AES key in either 128, 192, or 256-bits. This key is passed to the round function operation (CIPH(), which is AES in CBC-MAC mode) as seen below.

The tweak "serves much the same purpose that an initialization vector does for CBC mode" and provides variability to the ciphertext [13]. The tweak is very similar to the use of a salt value for securing storage of passwords; it helps avoid dictionary attacks and overcome the "fact that, with any deterministic encryption scheme, identical inputs do map to identical outputs" [3]. The tweak "does not need to be secret" [1] and "keeping the tweak secret need not provide any greater cryptographic strength" [13], it can be based in whole or in part on known information associated with the plaintext.

The input X is a numeral string and is a representation of a "set of two or more symbols called an *alphabet"* [1]. The *radix* is the base of the alphabet to be encrypted. For example, Social Security Numbers (decimal digits) would have a radix of 10, while the standard lowercase alphabet would have a radix of 26. Numeral strings are represented as lists of the set of non-negative integers and are converted between integers and numeral strings as necessary in the algorithms.

Note that "numbers are represented with *increasing* order of significance" [1] (i.e. the little-endian format) in the original BPS-BC specification [6]. The FF3-1 specification accounts for the little-endian format by providing utility functions to reverse numeral strings and byte arrays, as necessary, to conform to the BPS-BC format.

The FF3-1.Encrypt and FF3-1.Decrypt algorithms as specified in NIST 800-38G are provided below [1].

ALGORITHM 1: FF3-1.Encrypt [1]

```
1.  Let $u = \lceil n/2 \rceil; v = n - u.$
2.  Let $A = X[1..u]; B = X[u + 1..n]$
3.  Let $T_L = T[0..27] || 0^4$ and $T_R = T[32..55] || T[28..31] || 0^4$
4.  For $i$ from 0 to 7:
        i.   If $i$ is even, let $m = u$ and $W = T_R$, else let $m = v$ and $W = T_L$.
        ii.  Let $P = W \oplus [i]^4 || [NUM_{radix}(REV(B))]^{12}$
        iii. Let $S = REVB(CIPH_{REVB(K)} REVB(P))$
        iv.  Let $y = NUM(S)$
        v.   Let $c = (NUM_{radix}(REV(A)) + y) \bmod radix^m$
        vi.   Let $C = REV(STR^m_{radix}(c))$
        vii.  Let $A = B$
        viii. Let $B = C$
    5.  Return $A \,||\, B$
```

ALGORITHM 2: FF3-1.Decrypt [1]

```
    1.  Let $u = \lceil n/2 \rceil; v = n - u.$
    2.  Let $A = X[1..u]; B = X[u + 1..n]$
    3.  Let $T_L = T[0..27] \,||\, 0^4$ and $T_R = T[32..55] \,||\, T[28..31] \,||\, 0^4$
    4.  For $i$ from 0 to 7:
        i.    If $i$ is even, let $m = u$ and $W = T_R$, else let $m = v$ and $W = T_L$.
        ii.   Let $P = W \oplus [i]^4 \,||\, [NUM_{radix}(REV(A))]^{12}$
        iii.  Let $S = REVB(CIPH_{REVB(K)}REVB(P))$
        iv.   Let $y = NUM(S)$
        v.    Let $c = (NUM_{radix}(REV(B)) - y) \bmod radix^m$
        vi.   Let $C = REV(STR^m_{radix}(c))$
        vii.  Let B $= A$
        viii. Let A $= C$
    5.  Return $A \,||\, B$
```

The algorithms utilize a *Feistel structure*, which "consists of several iterations, called *rounds*, of reversible transformations" [1].

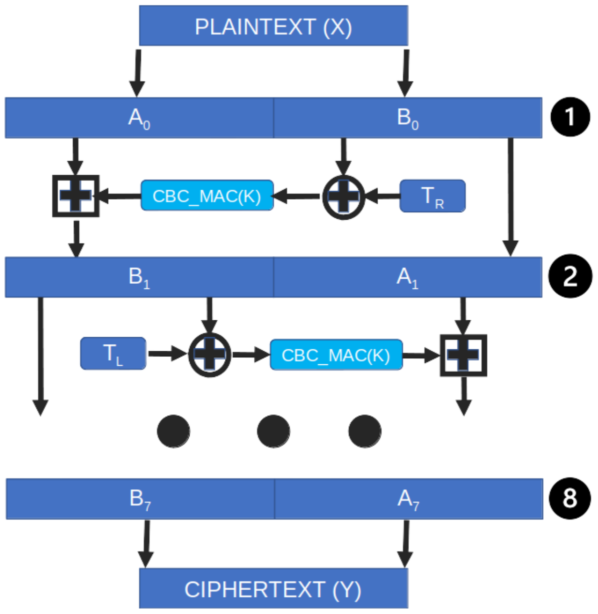

Figure 1: Representation of FF3-1 Feistel Structure

The plaintext (X) is split into two halves (A & B). The first iteration operates on the right half, as the B substring is XOR'd with the right half of the tweak value (split in Step 3 above). This output is fed into the CBC_MAC operation, operating under the key K. FF3-1 makes "extensive use of modular addition [6]", adding the results of the CBC_MAC operation with the substring B. Finally, the sides are switched, with the original B becoming the new A, and the modified A becoming the new B. FF3-1 utilizes eight Feistel rounds before returning the concatenated A and B substrings as the ciphertext (Y). Decryption utilizes a similar process but reverses the rounds and performs modular subtraction.

Feistel structures are well suited to forming the basis of format-preserving encryption algorithms as it "always engenders a reversible function – that is it works to make a blockcipher – no matter what you use for the round functions" [3].

3.2 THE FF3-1 IMPLEMENTATION AND INTEGRATION

For the implementation of FF3-1, the author first implemented the original version (FF3). This is due to the sample examples provided by NIST which provide intermediate values for both the encrypt and decrypt algorithms [20]. These samples were not updated for the FF3-1 algorithm.

The original FF3 implementation operates on numeral strings, and it was expected that utility code would convert the alphabet representation into the numeral string format before encrypt/decrypt operations. For base 10 numbers these can be represented with standard Python strings (*"0123456789"),* but if the radix is greater than base 10 they are formatted as a Python list: *["0", "1", "2", "3", "4", "5", "6", "7", "8", "9", "10", "11", "12", "13", "14"]*. The *_encrypt_numeral_string* and *_decrypt_numeral_string* methods are implementations of the original FF3 encrypt and decrypt algorithms working directly with numeral strings.

For the modified FF3-1 algorithm, the NIST ACVT [21] requires the support of capabilities, and effectively standardizes the supported characters for the alphabet. Valid alphabet characters must be between 2 and 64 characters in length [21]. The valid alphabet characters are restricted to decimals, lower and uppercase alphabets and the "+" and "/" characters and is synonymous to the Base64 character set (without the "=" padding character). This diverges from the original FF3 specification which allowed radix base values of up to 65536.

The *encrypt* and *decrypt* methods are FF3-1 wrappers to the original numeral_string methods, taking a plaintext in a string format (conforming to the alphabet), converting it to a numeral string, and converting the 56-bit tweak to a 64-bit tweak supported by FF3. Contrary to the FF3-1 algorithm spec, these methods do not take in a key value as input, as the key is specified in the construction of the object and cannot be changed without creating a new FF3 object. However, the tweak can (and should) be modified on each call to encrypt/decrypt, and to increase security should be unique for each plaintext/ciphertext pair.

The FF3-1 implementation methods are listed below. Note that per Python specification, an underscore denotes a private method that should not be called directly by API operations. Component functions are specified in Section 4.5 of the NIST 800-38G standard [1] and this implementation's methods closely follow the specification.

Table 3: FF3-1 Implementation Methods

Method	**Purpose**
__init__(self, radix, alphabet, key)	Initializes FF3 encryption object. Each FF3 object is limited to the same key, radix, and alphabet
_num_radix(self, X)	Takes numeral string X, returns the number that X represents in base radix when the numerals are valued in decreasing order of significance
_num(self, X)	Takes byte array X, returns an integer x valued in decreasing order of significance (i.e. big-endian)
_str_m_radix(self, m, x)	Given a non-negative integer less than radix^m, returns the representation of x as a string of m numerals in base radix, in big-endian

_rev(self, X)	Given a numeral string X, the numeral string that consists of the numerals of X in reverse order
_revb(self, X)	Given a bytearray X the byte array that consists of the bytes of X in reverse order
_check_numeral_string(self, X)	Checks the length of X and throws an error if not in range. Makes sure all elements of X are valid integers within the radix
_convert_tweak(self, T_56)	Takes a 56-bit tweak value and converts it to a 64-bit tweak value per Step 3 in Algorithm 9: FF3-1.Encrypt(K, T, X)
_plaintext_to_numeral_string(self, pt)	Takes a plaintext under a given alphabet and converts it to a numeral string. Validates the plaintext does not contain invalid characters.
_numeral_string_to_plaintext(self, ns)	Takes a plaintext under a given alphabet and converts it to a numeral string. Validates the plaintext does not contain invalid characters.
_encrypt_numeral_string(self, X, T)	Implements the FF3.Encrypt algorithm per NIST SP 800-38G. Takes a (plaintext) numeral string in base radix of length n, such that n is between minlen and maxlen. Takes tweak bit string T, such that len = 8 (64-bits). Returns (ciphertext) numeral string Y such that len(Y) = n
_decrypt_numeral_string(self, X, T)	Implements the FF3.Decrypt algorithm per NIST SP 800-38G. Takes a (ciphertext) numeral string in base radix of length n, such that n is between minlen and maxlen. Takes tweak bit string T, such that len = 8 (64-bits). Returns (plaintext) numeral string Y such that len(Y) = n
encrypt(self, pt, T_56)	Implements the FF3-1.Encrypt algorithm per NIST 800-38G rev. 1. Takes a plaintext and a 56-bit tweak, and returns a ciphertext which is the plaintext encrypted via the key and tweak, converted to a string
decrypt(self, ct, T_56)	Implements the FF3-1.Decrypt algorithm per NIST 800-38G rev. 1. Takes a ciphertext and a 56-bit tweak, and returns a plaintext which is the ciphertext decrypted via the key and tweak, converted to a string

3.3 PYCRYPTODOME INTEGRATION AND DELIVERABLES

The author provides the following contributions to the PyCryptodome cryptographic library. Note that while the name of the deliverable is FF3, this is in fact the FF3-1 implementation. Python does not allow hyphens within the naming convention for classes or functions, and thus the original name of the algorithm was retained. To the extent possible, the naming convention for variables followed in the FF3-1 specification [1] was retained.

Crypto/Cipher/ff3.py: Implementation of FF3-1 algorithm in PyCryptodome. It supports only the AES block cipher (in ECB mode) with a key in sizes of 128, 192, and 256 bits. The constructor takes as input the *alphabet*, *radix,* and *key*. The encrypt and decrypt methods take in a tweak and the plaintext or ciphertext (respectively) and returns the encrypted or decrypted result. While the key is locked with the instantiation of the FF3 object, the tweak may be changed on each subsequent encrypt and decrypt call.

The algorithm for FF3-1 calls for an approved block cipher in CBC-MAC mode. As only a single block is passed to the algorithm, in practice this is equivalent to ECB mode, which is implemented directly in PyCryptodome (with hardware acceleration support). Only AES is presently approved by NIST for the block cipher operation.

It should be noted that the NIST ACVP testing vectors limit the algorithm to an alphabet of 64 characters. The FF3-1 implementation is limited to sample cases which meet the NIST ACVP definitions.

Crypto/SelfTest/Cipher/test_ff3.py: Tests for the FF3-1 encrypt and decrypt functions. The tests include happy path testing to validate correct implementation against 450 NIST ACVT sample vectors [21], out of range testing for the key, radix, tweak, plaintexts and ciphertexts, valid alphabet character testing, and minimum and maximum input validation. The happy path tests are integrated with the *pycryptodome_test_vectors* project and will be skipped if the module is not installed.

PyCryptodome primarily utilizes the NIST CAVP test vectors. NIST is no longer publishing new CAVP test vectors and is transitioning to the Automated Cryptographic Validation Protocol. As this test support was not currently integrated within PyCryptodome, the test vectors have been added as an additional directory under **pycryptodome_test_vectors/Cipher/ACVP/FF3**

Doc/src/cipher/ff3.rst: Documentation (reStructuredText format) of the FF3-1 mode of operation. Includes the full API of the public encrypt and decrypt functions, examples, and notes for configuring the radix, alphabet, and tweak values.

Licensing: Care was taken to avoid other implementations of FF3 (and FF3-1) during the development process. Only the BPS specification [6] and NIST 800-38G [1] were consulted in developing the implementation, which is licensed for release under the BSD-2 clause [31].

The author has submitted the implementation as a pull request to the PyCryptodome project, where it awaits acceptance.

4 THREAT MODEL

Format-preserving encryption must provide strong cryptography. The NIST definition of strong cryptography is based on the security strength of the algorithm, “a number associated with the amount of work (i.e., the number of operations) that is required to break a cryptographic algorithm or system” [25]. For most symmetric cryptographic schemes and block ciphers, the most straightforward attack is a brute-force or key exhaustion attack; simply iterate over the key space until the correct key is found. For a block cipher with a 128-bit key space (such as AES) such an attack would require 2^{128} operations. NIST estimates the security strength of an algorithm “on many factors, including the attacker’s capabilities, the key lengths, the amount of data processed using the same key, and how closely keys are related” [25]. Per the NIST definition, the goal of FPE is to provide 128 bits of security strength, which is considered computationally infeasible through at least 2031.

The ideal security goal of format-preserving encryption is Pseudo-Random-Permutation (PRP) security. A formal definition of this goal is provided by Rogaway [3, 26]. An adversary with access to two oracles (one a random PRP generator, the other the FPE scheme) should have negligible advantage in distinguishing whether the oracle is returning the PRP or the FPE scheme [26]. Such an attack is referred to as a *distinguishing* attack. Distinguishing attacks against generic Feistel structures are presented by Patarin [15]. The authors of FF3 discount Patarin’s generic attacks in [15], as the number of queries exceed the aimed security goal and intended to leverage existing proofs of Feistel networks but noted “finding concrete bounds is still an open problem. Solving it would be very welcome since we are potentially manipulating very small plaintext” [6]. NIST SP 800-38G, originally published in 2016, placed domain restrictions (i.e. input restrictions) by setting the “radix minlen $\geq$ 100, in order to preclude a generic meet-in-the-middle attack on the Feistel structure” [27].

While strong PRP security is a stated goal for the FF1/FFX mode of operation [3], the authors of FF3/BPS did not consider the threat relevant in practice “since the technique only allows to distinguish several instances from our block cipher from a random keyed permutation family” [6]. Bellare et al. notes that while “a distinguishing attack aims to violate (tweakable) PRP security” they “have not been considered a significant threat in practice” as they do not cause any practical damage in applications of FPE [9].

Practical attacks against FPE schemes target message recovery. The first attacks against both FF1 and FF3 which result in message recovery are outlined by Bellare et al [9]. The attacks primarily target small domain sizes, and for 4-bit messages, the attackers fully recover the target message using 2^{31} examples" against FF3. Left-hand recovery and right-hand recovery attacks are combined and "when given three ciphertexts per tweak" can lead to recovery of the entire plaintext over small domains. The attacks on small domains are improved in [8, 10, 11, 14].

Durak and Vaudenay discovered a weakness in "bad domain separation" in the FF3 implementation, that effectively reduced the 8-round Feistel security of FF3 to 4 rounds [7]. The authors "developed a new generic known-plaintext attack to 4-round Feistel network" and showed that FF3 did not provide the desired 128-bit security properties. The attack manipulates a property of the tweak value in the function, where the tweak is XOR'd with the round function iterator. To prevent an attacker from being able to manipulate this field, the authors recommend a change in the tweak schedule [7].

To account for the attacks against small domain sizes, as well as the bad domain separation in the tweak for FF3-1, NIST issued a statement on the cryptanalysis of FF3 [12] and revised the 800-38G standard in 2019 [1]. The tweak modifications require "reordering some of (the tweak's) bits in a particular manner, and then forcing the bits in eight particular bit positions to be zero" [1]. In addition, the domain size is restricted for both FF1 and FF3 such that $\text{radix}^{\text{minlen}} \geq 1000000$. The use cases in Section 6 show how the domain restrictions affect the forms of input that can be used with FPE.

Note that under a single key and a single tweak value, an adversary with access to an encryption oracle can easily build a mapping of all plaintexts to all ciphertexts. For example, Social Security Numbers only have a billion possibilities, and by querying all known SSNs the adversary can then decrypt any ciphertext offline without access to the oracle. The authors of FF3-1 highly recommend a tweak value to protect against these dictionary attacks [6]. "Information that is available and statically associated with a plaintext" should be used "as a tweak for the plaintext" [1]. Under ideal scenarios, the tweak should be unique for each plaintext [1].

Finally, as Format-Preserving Encryption is a deterministic scheme, without the use of a tweak it cannot be considered IND-CPA (Indistinguishable vs. Chosen Plaintext Attack) [26]. As FPE does not include a message authentication code the scheme provides confidentiality only and cannot provide integrity of the ciphertext (INT-CTXT).

5 PERFORMANCE

To measure the performance of the implementation, the Python modules cProfile and timeit were used. The cProfile results include the number of calls, the total time spent in the given function, the time spent per call of a function, and the cumulative time spent in the function including subfunctions. The filename for each function is included, along with the line of the call. The cProfile results show that the utility functions add minimal overhead to the overall implementation.

Table 4: cProfile Results

ncalls	totime	percal	cumtime	Filename:lineno(function)
1	0.000	0.000	0.001	<string>:1(<module>)
8	0.000	0.000	0.000	FF3.py:103(_str_m_radix)
16	0.000	0.000	0.000	FF3.py:119(_rev)
24	0.000	0.000	0.000	FF3.py:129(_revb)
1	0.000	0.000	0.000	FF3.py:156(_convert_tweak)
1	0.000	0.000	0.000	FF3.py:183(_plaintext_to_numeral_string)
1	0.000	0.000	0.000	FF3.py:201(_numeral_string_to_plaintext
1	0.000	0.000	0.001	FF3.py:216(_encrypt_numeral_string
1	0.000	0.000	0.001	FF3.py:312(encrypt)
16	0.000	0.000	0.000	FF3.py:78(_num_radix)

```
8            0.000         0.000       0.000          FF3.py:92(_num)
```

The performance of the implementation was compared with the open-source implementation *Mysto-FF3*, which was written in Python and relies on (but does not provide integration with) PyCryptodome. 100,000 encryption operations were made with both implementations and the execution speed was timed with the timeit module. The Mysto implementation is approximately 10% faster than the author’s implementation. However, note that our implementation allows modifying the tweak value on each call to the encrypt and decrypt functions. When the test is performed when modifying the tweak on each call, the run-time of the two implementations is almost identical. The performance results show that our implementation performs well and can be used with real-world applications.

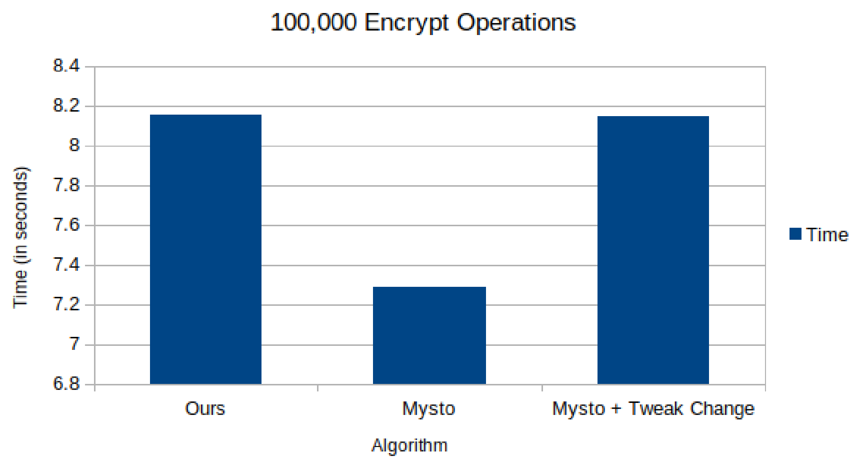

Figure 2: Performance results of 100,000 encrypt operations

6 USE CASES FOR FORMAT-PRESERVING ENCRYPTION

Use cases for format-preserving encryption include social security numbers, driver licenses, U.S. passport numbers, IP addresses, names, street addresses and email addresses, credit card and bank account numbers. Note that driver’s licenses, U.S. Passport numbers and taxpayer EINs are simply variations of the social security number use case. IP addresses will first require conversion from dotted decimal format to a standard integer. Certain IP addresses in the 0.0.0.0/8 block cannot be encrypted with FF3-1, but as this range is reserved it does not affect usability. Names and addresses have some usability restrictions, as the alphabet contains no native support for spaces, and shorter inputs either require padding or cannot be encrypted. The implementation does not handle conversions from typical format; any system utilizing our implementation is responsible for providing input in the appropriate format based on the alphabet of the FF3 object.

Table 1: FF3-1 Encryption Use Case Examples

Type	**Typical Format**	**FF3-1 Encrypt Input**	**Secure?**

Full Primary Account Number (PAN)	4111 1111 1111 1111	Alphabet: “0123456789” encrypt(4111111111111111, tweak)	Yes
Masked PAN (Standard)	**** ** 11 1111 ****	Alphabet: “0123456789” encrypt(111111, tweak)	Yes
Masked PAN (8 Digit BIN)	**** **** 1111 ****	Alphabet: “0123456789” encrypt(111111, tweak) Throws ValueError	No
Full Social Security Number	123-45-6789	Alphabet: “0123456789” encrypt(123456789, tweak)	Yes
Partial Social Security Number	123-45-****	Alphabet: “0123456789” encrypt(12345, tweak) Throws ValueError	No
Name	Joshua	Alphabet: “abcefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ” encrypt(“Joshua”, tweak)	Yes
Name	Holt	Alphabet: “abcefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ” encrypt(“Holt”, tweak) Throws ValueError	No
IP Address (v4)	192.168.0.1	Alphabet: “0123456789” encrypt(“3232235521”, tweak)	Yes
Bank Account Number	1234567891	Alphabet: “0123456789” encrypt(1234567891, tweak)	Yes
Routing Number	021000021	Alphabet: “0123456789” encrypt(“021000021”, tweak)	Yes

FF3-1 can help in pseudonymisation, “the processing of personal data in such a manner that the personal data can no longer be attributed to a specific data subject without the use of additional information” [16]. Care should be taken to ensure that pseudonymized data does not include additional fields which can be used to either infer or map the original sensitive data. Montjoye et al. show that “knowing the price of a (credit card) transaction increases the risk of reidentification by 22%, on average” [32]. Advice on complying with various privacy regulations such as the General Data Protection Regulation (GDPR) or California Consumer Privacy Act (CCPA) is beyond the scope of this paper.

Tweak values should be selected from information that is known and generally unique to the plaintext. This could be a non-sensitive identifier, the unmasked portions of an input, or simply a random value if supported. The authors of BPS-BC “suggest to apply a truncated hash function on the tweak input data” to meet the strict size limitations of the tweak input [6].

Note that format-preserving encryption can cause issues with Data Loss Prevention (DLP) security controls, as the DLP solution cannot differentiate between an FPE ciphertext and an unprotected plaintext. Integrity checks, such as the Luhn checksum for credit card numbers or the mod 10 check for routing numbers, will not be successful without special consideration.

7 CONCLUSION AND FUTURE WORK

Format-preserving encryption adds an additional tool to the cryptographer's toolbox. By providing a reference implementation of FF3-1 integrated with PyCryptodome, developers can rely on the standard API and the provided use cases to avoid rolling their own crypto.

Future work includes closely watching the NIST 800-38G Rev. 1 draft status [1]. More changes to FF3-1 are likely based on recent cryptanalysis, and NIST may certify additional FPE algorithms such as FF4. Additional tools, scaffolding, and functionality can be built around the implementation, such as providing encryption-as-a-service, REST API microservices, cloud-based implementation functions (such as AWS Lambda) and extensions to popular database engines (such as Postgresql). Finally, if the implementation is not accepted into PyCryptodome due to inactivity with the project, the author may elect to fork the project and continue maintenance.

8 ACKNOWLEDGMENTS

The author would like to thank Mustaque Ahamad, Karl Grindal, and Nate Lewis for their feedback and direction during the capstone project journey.

REFERENCES

[1] Morris Dworkin. 2019. Draft NIST Special Publication 800-38G Revision 1 - Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption. Retrieved from https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-38Gr1-draft.pdf

[2] Bellare, Ristenpart, Rogaway, Stegers. 2009. Format-Preserving Encryption. Retrieved from https://eprint.iacr.org/2009/251.pdf

[3] Phillip Rogaway. 2010. A Synopsis of Format-Preserving Encryption. Retrieved from https://www.cs.ucdavis.edu/~rogaway/papers/synopsis.pdf

[4] Mihir Bellare, Phillip Rogaway, Terence Spies. 2010 – The FFX Mode of Operation for Format-Preserving Encryption. Retrieved from https://csrc.nist.gov/CSRC/media/Projects/Block-Cipher-Techniques/documents/BCM/proposed-modes/ffx/ffx-spec.pdf

[5] Mihir Bellare, Phillip Rogaway, Terence Spies. 2010. Addendum to "The FFX Mode of Operation for Format-Preserving Encryption" A parameter collection for enciphering strings of arbitrary radix and length. Retrieved from https://csrc.nist.gov/CSRC/media/Projects/Block-Cipher-Techniques/documents/BCM/proposed-modes/ffx/ffx-spec2.pdf

[6] Eric Brier, Thomas Peyrin and Jacques Stern. 2010. BPS: a Format-Preserving Encryption Proposal. Retrieved from http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.694.918&rep=rep1&type=pdf

[7] F. Betul Durak and Serge Vaudenay. 2017. Breaking The FF3 Format-Preserving Encryption Standard Over Small Domains. Retrieved from https://eprint.iacr.org/2017/521.pdf

[8] Viet Tung Hoang, Stefano Tessaro, and Ni Trieu. 2018. The Curse of Small Domains: New Attacks on Format-Preserving Encryption. Retrieved from https://eprint.iacr.org/2018/556.pdf

[9] Mihir Bellare, Viet Tung Hoang, Stefano Tessaro. 2016. Message-Recovery Attacks on Feistel-Based Format Preserving Encryption. Retrieved from https://dl.acm.org/doi/pdf/10.1145/2976749.2978390

[10] Orr Dunkelman, Abhishek Kumar, Eran Lambooij, Somitra Kumar Sanadhya. 2020. Cryptanalysis of Feistel-Based Format-Preserving Encryption. Retrieved from https://eprint.iacr.org/2020/1311.pdf

[11] Tim Beyne. 2021. Linear Cryptanalysis of FF3-1 and FEA. Retrieved from https://www.esat.kuleuven.be/cosic/publications/article-3384.pdf

[12] NIST. 2017. Recent Cryptanalysis of FF3. Retrieved from https://csrc.nist.gov/news/2017/recent-cryptanalysis-of-ff3

[13] Moses Liskov, Ronald L. Rivest, and David Wagner. Tweakable Block Ciphers. 2002. Retrieved from https://people.csail.mit.edu/rivest/pubs/LRW02.pdf

[14] Ohad Amon, Orr Dunkelman, Nathan Keller, Eyal Ronen, and Adi Shamir. 2021. Three Third Generation Attacks on the Format Preserving Encryption Scheme FF3. Retrieved from https://eprint.iacr.org/2021/335.pdf

[15] Jacques Patarin. 2001. Generic Attacks on Feistel Schemes. Retrieved from https://eprint.iacr.org/2008/036.pdf

[16] European Union Agency for Cybersecurity. Pseudonymisation techniques and best practices - Recommendations on shaping technology according to data protection and privacy provisions. Retrieved from https://www.enisa.europa.eu/publications/pseudonymisation-techniques-and-best-practices/at_download/fullReport

[17] Acar, Backes, Fahl, Garfinkel, Kim, Mazurek, Stransky. 2017. Comparing the Usability of Cryptographic APIs. Retrieved from https://www.cl.cam.ac.uk/~rja14/shb17/fahl.pdf

[18] PCI Security Standards Council. Tokenization Product Security Guidelines. Retrieved from https://www.pcisecuritystandards.org/documents/Tokenization_Product_Security_Guidelines.pdf

[19] PCI Security Standards Council. Information Supplement: PCI DSS Tokenization Guidelines. Retrieved from https://www.pcisecuritystandards.org/documents/Tokenization_Guidelines_Info_Supplement.pdf

[20] NIST. Block Cipher Modes of Operation: FF3 Method for Format-Preserving Encryption. Retrieved from https://csrc.nist.gov/CSRC/media/Projects/Cryptographic-Standards-and-Guidelines/documents/examples/FF3samples.pdf

[21] NIST. Automated Cryptographic Validation Testing (ACVT). https://csrc.nist.gov/Projects/Automated-Cryptographic-Validation-Testing

[22] NIST. Cryptographic Algorithm Validation Program (CAVP). https://csrc.nist.gov/projects/cryptographic-algorithm-validation-program

[23] Matthew Pauker, Terence Spies, Luther Martin. 2006. Data processing systems with format-preserving encryption and decryption engines. (June 2006). US Patent No. 7864952 B2, Filed Dec. 6, 2006, Issued Jan. 4, 2011.

[24] Voltage Security. REVISED LETTER OF ASSURANCE FOR ESSENTIAL PATENT CLAIMS FFX Mode of Operation for Format-Preserving Encryption. Retrieved April 2, 2013. https://csrc.nist.gov/csrc/media/projects/block-cipher-techniques/documents/bcm/proposed-modes/ffx/ffx-voltage-ip.pdf

[25] NIST. Special Publication 800-57 Part 1 Revision 5: Recommendation for Key Management: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf

[26] Bellare, Rogaway. Introduction to Modern Cryptography. 2005. Retrieved from https://web.cs.ucdavis.edu/~rogaway/classes/227/spring05/book/main.pdf

[27] Morris Dworkin. 2016. NIST Special Publication 800-38G - Recommendation for Block Cipher Modes of Operation: Methods for Format-Preserving Encryption. Retrieved from https://csrc.nist.gov/publications/detail/sp/800-38g/final

[28] Bellare, Hoang. 2017. Identity-Based Format-Preserving Encryption. Retrieved from https://eprint.iacr.org/2017/877.pdf

[29] Joachim Vance, Mihir Bellare. 2014. An Extension of the FF2 FPE Scheme. Retrieved from https://csrc.nist.gov/CSRC/media/Projects/Block-Cipher-Techniques/documents/BCM/proposed-modes/dff/dff-ff2-fpe-scheme-update.pdf

[30] Wonyoung Jang, Sun-Young Lee. 2020. A Format-preserving encryption FF1, FF3-1 Using Lightweight Block Ciphers LEA and SPECK. Retrieved from https://dl.acm.org/doi/pdf/10.1145/3341105.3373953

[31] The 2-Clause BSD License. Retrieved from https://opensource.org/licenses/BSD-2-Clause

[32] Montjoye, Radaelli, Singh, Pentland. 2016. Unique in the shopping mall: On the reidentifiability of credit card metadata. Retrieved from https://www.science.org/doi/10.1126/science.1256297

A APPENDICES

A.1 Industry Implementations of FPE

[1] ANSI X9.124-2-2018: Symmetric Key Cryptography for the Financial Services Industry – Format Preserving Encryption – Part 2: Key Stream with Counter Mode. Retrieved from https://webstore.ansi.org/standards/ascx9/ansix91242018

[2] IBM Systems cryptographic HSMs. Format perserving encryption. Retrived from https://www.ibm.com/docs/en/linux-on-systems?topic=services-format-preserving-encryption and https://www.ibm.com/security/cryptocards

[3] Hashicorp. Vault Enterprise Advanced Data Protection Module. Retrieved from https://www.hashicorp.com/products/vault/transform

[4] Micro Focus. Voltage SecureData. Retrieved from https://www.microfocus.com/media/data-sheet/voltage_securedata_ds.pdf

[5] Bluefin. ShieldConex. Retrieved from https://www.bluefin.com/products/shieldconex/

A.2 Open-Source Implementations of FPE

[1] Format-preserving implementation in Go. Retrieved from https://github.com/capitalone/fpe

[2] Mysto Format-Preserving Encryption implementations. Retrieved from https://github.com/mysto

[3] LibFFX – Python implementation of the FFX mode of operation for Format-Preserving Encryption. Retrieved from https://github.com/kpdyer/libffx

[4] PyFFX – Python implementation of Format-preserving Feistel-based encryption (FFX). Retrieved from https://github.com/emulbreh/pyffx

[5] BouncyCastle FPEEngine. Retrieved from https://www.bouncycastle.org/docs/docs1.8on/index.html

A.3 FF3-1 Python Implementation

Github Repository: https://github.com/Dregnus/pycryptodome

Pull Request: https://github.com/Legrandin/pycryptodome/pull/601

Source code for the FF3-1 implementation is provided below, stripped of comments and docstrings for brevity. Author retains all rights under BSD-2 clause license.

```python
from __future__ import absolute_import
import math
from Crypto.Cipher import AES
from Crypto.Random import get_random_bytes
class FF3:
    def __init__(self, radix, alphabet, key):
        self.radix = radix
        if (self.radix < 2) or (self.radix > 64):
            raise RadixOutOfRangeError('Radix must be between 2 and 64')
        self.alphabet = alphabet
        if (len(self.alphabet) < 2) or (len(self.alphabet) > 64):
            raise AlphabetOutOfRangeError("Alphabet range between 2 and 64")
        if not (self.alphabet.isalnum()):
            if "+/" not in self.alphabet:
                raise AlphabetValueError("Alphabets must contain numbers \
                    and upper and lower case letters, '+ and '/)")
        if len(self.alphabet) != len(set(self.alphabet)):
            raise AlphabetValueError("All alphabet values must be unique")
        self.key = key
        self.ciph = AES.new(self._revb(self.key), AES.MODE_ECB)
        self.minlen = math.ceil(math.log(1000000) / math.log(radix))
        self.maxlen = 2 * math.floor(math.log(2 ** 96, radix))
    def _num_radix(self, X):
        x = 0
        for i in range(0, len(X)):
            x = x * self.radix + int(X[i])
        return x
    def _num(self, X):
        return int.from_bytes(X, "big")
    def _str_m_radix(self, m, x):
        X = []
        for i in range(0, m):
            X.insert(0, str(x % self.radix))
            x = x // self.radix
        return X
    def _rev(self, X):
        return X[::-1]
    def _revb(self, X):
```

```
    return X[::-1]
def _check_numeral_string(self, X):
    if not (self.minlen <= len(X) <= self.maxlen):
        raise ValueError('Length must be between {} and {}'
                         .format(self.minlen, self.maxlen))
    for i in X:
        try:
            num = int(i)
            if not (0 <= num < self.radix):
                raise ValueError('Element must be integer within base {}'
                                 .format(self.radix))
        except ValueError:
            raise ValueError('Element must be an integer within base {}'
                             .format(self.radix))
def _convert_tweak(self, T_56):
    if type(T_56) is not (bytes or bytearray):
        raise TypeError('Tweak must be bytes or bytearray')
    if len(T_56) != 7:
        raise ValueError('Tweak must be 7 bytes or 56 bits in length')
    t_l = int.from_bytes(T_56[0:4], 'big')
    t_l = (t_l >> 4) << 4
    t_l = t_l.to_bytes(4, 'big')
    t_r = int.from_bytes(T_56[4:], 'big')
    t_r = t_r << 8
    t_r = t_r | ((T_56[3] & 15) << 4)
    t_r = t_r.to_bytes(4, 'big')
    return t_l + t_r
def _plaintext_to_numeral_string(self, pt):
    X = []
    for char in pt:
        try:
            X.append(self.alphabet.index(char))
        except ValueError:
            raise AlphabetValueError("Plaintext element {} not in \
                alphabet".format(char))
    return X
def _numeral_string_to_plaintext(self, numeral_string):
    pt = []
    for number in numeral_string:
        pt.append(self.alphabet[int(number)])
    pt = ''.join(pt)
    return pt
def _encrypt_numeral_string(self, X, T):
    n = len(X)
    u = math.ceil(n / 2)
    v = n - u
    A, B = X[:u], X[u:n]
    T_L, T_R = T[:4], T[4:]
    for i in range(0, 8):
        if (i % 2 == 0):
            m, w = u, T_R
        else:
            m, w = v, T_L
        w = bytearray(w)
        w[3] = w[3] ^ i
        P = w + self._num_radix(self._revb(B)).to_bytes(12,
                                                        byteorder='big')
        S = self._revb(self.ciph.encrypt(self._revb(P)))
        y = self._num(S)
        c = (self._num_radix(self._rev(A)) + y) % (pow(self.radix, m))
        C = self._rev(self._str_m_radix(m, c))
        A = B
        B = C
    return A + B
def _decrypt_numeral_string(self, X, T):
    n = len(X)
    u = math.ceil(n / 2)
    v = n - u
    A, B = X[:u], X[u:n]
    T_L, T_R = T[:4], T[4:]
    for i in range(7, -1, -1):
        if (i % 2 == 0):
            m, w = u, T_R
        else:
            m, w = v, T_L
        w = bytearray(w)
        w[3] = w[3] ^ i
        P = w + self._num_radix(self._revb(A)).to_bytes(12,
                                                        byteorder='big')
        S = self._revb(self.ciph.encrypt(self._revb(P)))
        y = self._num(S)
        c = (self._num_radix(self._rev(B)) - y) % (pow(self.radix, m))
```

```
                C = self._rev(self._str_m_radix(m, c))
                B = A
                A = C
            return A + B
        def encrypt(self, pt, T_56):
            if not (self.minlen <= len(pt) <= self.maxlen):
                raise ValueError("Length of pt must be between \
                    {} and {}".format(self.minlen, self.maxlen))
            X = self._plaintext_to_numeral_string(pt)
            T_64 = self._convert_tweak(T_56)
            Y = self._encrypt_numeral_string(X, T_64)
            return self._numeral_string_to_plaintext(Y)
        def decrypt(self, ct, T_56):
            if not (self.minlen <= len(ct) <= self.maxlen):
                raise ValueError("Length of ct must be between \
                    {} and {}".format(self.minlen, self.maxlen))
            X = self._plaintext_to_numeral_string(ct)
            T_64 = self._convert_tweak(T_56)
            Y = self._decrypt_numeral_string(X, T_64)
            return self._numeral_string_to_plaintext(Y)
    class RadixOutOfRangeError(ValueError):
        pass
    class AlphabetOutOfRangeError(ValueError):
        pass
    class AlphabetValueError(ValueError):
        pass
```

A.4 FF3-1 PyCryptodome Documentation

FF3 (Format Preserving Encryption) is a a method of encryption which encrypts a plaintext into a ciphertext while preserving the format of the plaintext. PyCryptodome implements FF3-1 as outlined in NIST 800-38G NIST .

Format Preserving Encryption is useful for legacy systems and other situations where sensitive data must be protected, but the format and the length must be retained. Common examples include Social Security Numbers (SSNs) and credit card numbers.

FF3 uses the AES block cipher under the hood in CBC-MAC mode, and supports keys lengths of 128, 192, or 256 bits long.

Format Preserving Encryption has a few unique properties which are required to successfully use the algorithm:

1. **Alphabet**: Alphabets represent the valid characters that can appear in a plaintext. For SSNs and credit cards, which can contain only digits, the alphabet would be "0123456789". NIST ACVP defines an alphabet as a minimum of two characters, and a maximum of 64 (all numbers and upper and lower case letters, additionally "+" and "/").

2. **Radix**: The radix is simply the length of the alphabet, and represents the number base. For example, SSNs are decimal digits and are in base 10.

3. **Tweak**: A tweak is a non-secret value that can be used to change part of the key. Tweaks are necessary in Format Preserving Encryption because the domain of ciphertexts can be relatively low. FF3-1 tweaks must be 7 bytes in length. Any information that is available and associated with a plaintext can be used as a tweak. It's very similar to a salt value in that it doesn't need to be secret, but should be unique. Tweaks should be used whenever possible to limit guessing attacks.

FF3-1 example:

```
>>> from Crypto.Cipher.FF3 import FF3
>>> from Crypto.Random import get_random_bytes
>>>
>>> alphabet = "0123456789"
>>> radix = len(alphabet)
>>> key = get_random_bytes(16)
>>> fpe = FF3(radix, alphabet, key)
```

You can encrypt a plaintext by passing the plaintext and a tweak to the encrypt() method:

```
>>> tweak = get_random_bytes(7)
>>> pt = "123456789"
>>> ct = fpe.encrypt(pt, tweak)
>>> print(ct)
930076983
```

You can decrypt a ciphertext by passing the ciphertext and a tweak to the decrypt() method:

```
>>> pt = fpe.decrypt(ct, tweak)
>>> print(pt)
123456789
```

FPE is deterministic, and the same plaintext and tweak values will provide the same ciphertext. However, modifying the tweak value will change the associated ciphertext:

```
>>> ct = fpe.encrypt(pt, tweak)
>>> print(ct)
930076983
>>> tweak = get_random_bytes(7)
>>> ct = fpe.encrypt(pt, tweak)
>>> print(ct)
138680525
>>> pt = fpe.decrypt(ct, tweak)
>>> print(pt)
123456789
```

Note that NIST also defines FF1, which has patent claims and is not implemented by PyCryptodome.